Resource Management Interface Specs
Release v. 3.0.3
29 JUN 2004

Scott Jackson
Brett Bode
David Jackson
Kevin Walker

Scalable Systems Software Resource Management and Accounting Protocol (SSSRMAP) Wire Protocol

Status of this Memo

This is a specification defining a wire level protocol used between Scalable Systems Software components. It is intended that this specification will continue to evolve as these interfaces are implemented and thoroughly tested by time and experience.

Abstract

This document is a specification describing a connection-oriented XML-based application layer client-server protocol for the interaction of resource management and accounting software components developed as part of the Scalable Systems Software Center. The SSSRMAP Wire Protocol defines a framing protocol that includes provisions for security. The protocol is specified in XML Schema Definition and rides on the HTTP protocol.

Table of Contents

1 Introduction 2
2 Conventions Used in this Document 2
2.1 Keywords 3
2.2 XML Case Conventions 3
2.3 Schema Definitions 3
3 Encoding 3
3.1.1 Schema Header and Namespaces 3
3.1.2 The *Envelope* Element 4
3.1.3 The *Body* Element 4
4 Transport Layer 5
5 Framing 5
5.1 Message Header Requirements 5
5.2 Message Chunk Format 6
5.3 Reply Header Requirements 6
5.4 Reply Chunk Format 6
5.5 Message and Reply Tail Requirements and Multiple Chunks 6
5.6 Examples 7
5.6.1 Sample SSSRMAP Message Embedded in HTTP Request 7
5.6.2 Sample SSSRMAP Reply Embedded in HTTP Response 7
6 Asynchrony 7

7 Security 8
7.1 Security Token 8
7.1.1 The *SecurityToken* Element 8
7.1.2 Security Token Types 9
7.1.2.1 Symmetric Key 9
7.1.2.2 Asymmetric Key 9
7.1.2.3 Password 10
7.1.2.4 Cleartext 10
7.1.2.5 Kerberos 10
7.1.2.6 GSI (X.509) 11
7.1.3 Example 11
7.2 Authentication 11
7.2.1 The *Signature* Element 12
7.2.2 The *DigestValue* Element 12
7.2.3 The *SignatureValue* Element 13
7.2.4 Signature Example 13
7.3 Confidentiality 14
7.3.1 The *EncryptedData* Element 15
7.3.2 The *EncryptedKey* Element 15
7.3.3 The *CipherValue* Element 16
7.3.4 Encryption Example 17
8 Acknowledgements 18
9 References 18

1 Introduction

A major objective of the Scalable Systems Software [SSS] Center is to create a scalable and modular infrastructure for resource management and accounting on terascale clusters including resource scheduling, grid-scheduling, node daemon support, comprehensive usage accounting and user interfaces emphasizing portability to terascale vendor operating systems. Existing resource management and accounting components feature disparate APIs (Application Programming Interfaces) requiring various forms of application coding to interact with other components.

This document proposes a wire level protocol expressed in an XML envelope to be considered as the foundation of a standard for communications between and among resource management and accounting software components. Individual components additionally need to define the particular XML binding necessary to represent the message format for communicating with the component.

2 Conventions Used in this Document

2.1 Keywords

The keywords “MUST”, “MUST NOT”, “REQUIIRED”, “SHALL”, “SHALL NOT”, “SHOULD”, “RECOMMENDED”, “MAY”, and “OPTIONAL” in this document are to be interpreted as described in RFC2119 [RFC2119].

2.2 XML Case Conventions

In order to enforce a consistent capitalization and naming convention across all SSSRMAP specifications “Upper Camel Case” (UCC) and “Lower Camel Case” (LCC) Capitalization styles shall be used. UCC style capitalizes the first character of each word and compounds the name. LCC style capitalizes the first character of each word except the first word. [XML_CONV][FED_XML]

1. SSSRMAP XML Schema and XML instance documents SHALL use the following conventions:
    - Element names SHALL be in UCC convention (example: <UpperCamelCaseElement/>.
    - Attribute names SHALL be in LCC convention (example: <UpperCamelCaseElement lowerCamelCaseAttribute=”Whatever”/>.
2. General rules for all names are:
    - Acronyms SHOULD be avoided, but in cases where they are used, the capitalization SHALL remain (example: XMLSignature).
    - Underscores (_), periods (.) and dashes (-) MUST NOT be used (example: use JobId instead of JOB.ID, Job_ID or job-id).

2.3 Schema Definitions

SSSRMAP Schema Definitions appear like this

In case of disagreement between the schema file and this specification, the schema file takes precedence.

3 Encoding

Encoding tells how a message is represented when exchanged. SSSRMAP data exchange messages SHALL be defined in terms of XML schema [XML_SCHEMA].

3.1.1 Schema Header and Namespaces

The header of the schema definition is as follows:

```
<?xml version="1.0" encoding="UTF-8"?>

<schema
  xmlns="http://www.w3.org/2001/XMLSchema"
  xmlns:sssrmap="http://www.scidac.org/ScalableSystems/SSSRMAP"
  targetNamespace="http://www.scidac.org/ScalableSystems/SSSRMAP"
  elementFormDefault="qualified">
```

3.1.2 The *Envelope* Element

SSSRMAP messages and replies are encapsulated in the *Envelope* element. There are two possibilities for the contents of this element. If the contents are unencrypted, this element MUST contain a *Body* element and MAY contain a *Signature* element (refer to the section on Security). If the contents are encrypted, this element MUST contain exactly one *EncryptedData* element (refer to the section on Security). The *Envelope* element MAY contain namespace and other xsd-specific information necessary to validate the document against the schema. In addition, it MAY have any of the following attributes which may serve as processing clues to the parser:

- *type* – A message type providing a hint as to the body contents such as "Request" or "Notification"
- *component* – A component type such as "QueueManager" or "LocalScheduler"
- *name* – A component name such as "OpenPBS" or "Maui"
- *version* – A component version such as "2.2p12" or "3.2.2"

```
<complexType name="EnvelopeType">
  <choice minOccurs="1" maxOccurs="1">
    <choice minOccurs="1" maxOccurs="2">
      <element ref="sssrmap:Signature" minOccurs="0" maxOccurs="1"/>
      <element ref="sssrmap:Body" minOccurs="1" maxOccurs="1"/>
    </choice>
    <element ref="sssrmap:EncryptedData" minOccurs="1" maxOccurs="1"/>
  </choice>
  <attribute name="type" type="string" use="optional"/>
  <attribute name="component" type="string" use="optional"/>
  <attribute name="name" type="string" use="optional"/>
  <attribute name="version" type="string" use="optional"/>
</complexType>

<element name="Envelope" type="sssrmap:EnvelopeType"/>
```

3.1.3 The *Body* Element

- SSSRMAP messages and replies are encapsulated in the *Body* element. This element MUST contain exactly one *Request* or *Response* element.

```
<complexType name="BodyType">
        <choice minOccurs="1" maxOccurs="1">
<element ref="sssrmap:Request" minOccurs="0" maxOccurs="1"/>              <element
ref="sssrmap:Response" minOccurs="0" maxOccurs="1"/>
    <any minOccurs="0" maxOccurs="1" namespace="##other"/>
  </choice>
</complexType>

<element name="Body" type="sssrmap:BodyType"/>
```

4 Transport Layer

This protocol will be built over the connection-oriented reliable transport layer TCP/IP. Support for other transport layers could also be considered, but native support for TCP/IP can be found on most terascale clusters and automatically handles issues such as reliability and connectionfullness for the application developer implementing the SSSRMAP protocol.

5 Framing

Framing specifies how the beginning and ending of each message is delimited. Given that the encoding will be expressed as one or more XML documents, clients and servers need to know when an XML document has been fully read in order to be parsed and acted upon.

SSSRMAP uses the HTTP 1.1 [HTTP] protocol for framing. HTTP uses a byte-counting mechanism to delimit the message segments. HTTP chunked encoding is used. This allows for optional support for batched messages, large message segmentation and persistent connections.

5.1 Message Header Requirements

The HTTP request line (first line of the HTTP request header) begins with POST and is followed by a URI and the version of the HTTP protocol that the client understands. The URI MUST consist of a single slash, followed by the protocol name in uppercase and a major version number (i.e. /SSSRMAP3).

The Content-Type must be specified as test/xml. Charset may be optionally specified and defaults to US-ASCII. It is recommended that charset be specified as “utf-8” for maximum interoperability.

The Transfer-Encoding must be specified as chunked. The Content-Length must NOT be specified as the chunk size is specified in the message chunk.

Other properties such as User-Agent, Host and Date are strictly optional.

5.2 Message Chunk Format

A message chunk consists of a chunk size in hexadecimal format (whose value is the number of bytes in the XML message not including the chunk size and delimiter) delimited by a CR/LF "\r\n" and followed by the message payload in XML that consists of a single XML document having a root element of *Envelope*.

5.3 Reply Header Requirements

The HTTP response line (first line of the HTTP response header) begins with HTTP and a version number, followed by a numeric code and a message indicating what sort of response is made. These response codes and messages indicate the status of the entire response and are as defined by the HTTP standard. The most common response is 200 OK, indicating that the message was received and an appropriate response is being returned.

The Content-Type must be specified as text/xml. Charset may be optionally specified and defaults to US-ASCII. It is recommended that charset be specified as "utf-8" for maximum interoperability.

The Transfer-Encoding MUST be specified as chunked. The Content-Length must NOT be specified.

Other properties such as Server, Host and Date are strictly optional.

5.4 Reply Chunk Format

A reply chunk consists of a chunk size in hexadecimal format (whose value is the number of bytes in the XML reply not including the chunk size and delimiter) delimited by a CR/LF "\r\n" and followed by the reply payload in XML that consists of a single XML document having a root element of *Envelope*.

5.5 Message and Reply Tail Requirements and Multiple Chunks

This specification only requires that single chunks be supported. A server may optionally be configured to handle requests with persistent connections (multiple

chunks). It will be the responsibility of clients to know whether a particular server supports this additional functionality. After all chunks have been sent, a connection is terminated by sending a zero followed by a carriage return-linefeed combination (0\r\n) and closing the connection.

5.6 Examples

5.6.1 Sample SSSRMAP Message Embedded in HTTP Request

```
POST /SSSRMAP3 HTTP/1.1\r\n
Content-Type: text/xml; charset=”utf-8”\r\n
Transfer-Encoding: chunked\r\n
\r\n
9A\r\n
<Envelope …/>
0\r\n
```

5.6.2 Sample SSSRMAP Reply Embedded in HTTP Response

```
HTTP/1.1 200 OK\r\n
Content-Type: text/xml; charset=”utf-8”\r\n
Transfer-Encoding: chunked\r\n
\r\n
2B4\r\n
<Envelope …/>
0\r\n
```

6 Asynchrony

Asynchrony (or multiplexing) allows for the handling of independent exchanges over the same connection. A widely-implemented approach is to allow pipelining (or boxcarring) by aggregating requests or responses within the body of the message or via persistent connections and chunking in HTTP 1.1. Pipelining helps reduce network latency by allowing a client to make multiple requests of a server, but requires the requests to be processed serially [RFC3117]. Parallelism could be employed to further reduce server latency by allowing multiple requests to be processed in parallel by multi-threaded applications.

Segmentation may become necessary if the messages are larger than the available window. With support for segmentation, the octet-counting requirement that you need to know the length of the whole message before sending it can be relegated to the segment level – and you can start sending segments before the whole message is available. Segmentation is facilitated via “chunking” in HTTP 1.1.

The current SSSRMAP strategy supports only a single request or response within the Body element. A server may optionally support persistent connections from a client via HTTP chunking. Segmentation of large responses is also optionally supported via HTTP chunking. Later versions of the protocol could allow pipelined requests and responses in a single Body element.

7 Security

SSSRMAP security features include capabilities for integrity, authentication, confidentiality, and non-repudiation. The absence or presence of the various security features depend upon the type of security token used and the protection methods you choose to specify in the request.

For compatibility reasons, SSSRMAP specifies six supported security token types. Extensibility features are included allowing an implementation to use alternate security algorithms and security tokens. It is also possible for an implementation to ignore security features if it is deemed nonessential for the component. However, it is highly RECOMMENDED that an implementation support at least the default security token type in both authentication and encryption.

7.1 Security Token

A security token may be included in either the Signature block, and/or in the EncryptedData block (both described later) as an implicit or explicit cryptographic key. If this element is omitted, the security token is assumed to be a secret key shared between the client and the server.

7.1.1 The *SecurityToken* Element

This element is of type String. If the security token conveys an explicit key, this element's content is the value of the key. If the key is natively expressed in a binary form, it must be converted to base64 encoding as defined in XML Digital Signatures ("http://www.w3.org/2000/09/xmldsig#base64"). If the type is not specified, it is assumed to be of type "Symmetric".

It may have any of the following optional attributes:

- *type* – the type of security token (described subsequently)
    - A *type* attribute of "Symmetric" specifies a shared secret key between the client and server. This is the default.

    - A *type* attribute of “Asymmetric” specifies the use of public private key pairs between the client and server.
    - A *type* attribute of “Password” encrypts and authenticates with a user password known to both the client and server.
    - A *type* attribute of “Cleartext” allows the passing of a cleartext username and password and depends on the use of a secure transport (such as SSL or IPSec).
    - A *type* attribute of “Kerberos5” specifies a kerberos token.
    - A *type* attribute of “X509v3” specifies an X.509 certificate.
- *name* – the name of the security token which serves as an identifier for the actor making the request (useful when the key is a password, or when the key value is implicit as when a public key is named but not included)

```
<complexType name="SecurityTokenType" mixed=”true”>
  <simpleContent>
    <extension base="string">
      <attribute name="type" type="string" use="optional"/>
      <attribute name="name" type="string" use="optional"/>
    </extension>
  </simpleContent>
</complexType>

<element name=”SecurityToken" type="sssrmap:SecurityTokenType"/>
```

7.1.2 Security Token Types

SSSRMAP defines six standard security token types:

7.1.2.1 Symmetric Key

The default security token specifies the use of a shared secret key. The secret key is up to 128-bits long and known by both client and server. When using a symmetric key as a security token, it is not necessary to specify the *type* attribute with value “Symmetric” because this is assumed when the attribute is absent. The *name* attribute may be used to specify which key to use if there are multiple symmetric keys. If the user provides a password to be sent to the server for authentication, then the password is encrypted with the secret key using a default method=”kw-tripledes” (XML ENCRYPTION http://www.w3.org/2001/04/xmlenc#kw-tripledes), base64 encoded and included as the string content of the *SecurityToken* element. If the client authenticated the user, then the *SecurityToken* element is empty .The same symmetric key is used in both authentication and encryption.

7.1.2.2 Asymmetric Key

Public and private key pairs can be used to provide non-repudiation of the client (or server). The client and the server must each have their own asymmetric key pairs. This mode is indicated by specifying the *type* attribute as “Asymmetric”. The *name* attribute should be specified indicating the actor issuing the request. If the user provides a password to be sent to the server for authentication, then the password is encrypted with the server’s public key using a default method=”rsa-1_5” (XML ENCRYPTION http://www.w3.org/2001/04/xmlenc#rsa-1_5), base64 encoded and included as the string content of the *SecurityToken* element. If the client authenticated the user, then the *SecurityToken* element is empty .The sender’s private key is used in authentication (signing) while the recipient’s public key is used for encryption.

7.1.2.3 Password

This mode allows for a username password combination to be used under the assumption that the server also knows the password for the user. This security token type is indicated by specifying a value of “Password” for the *type* attribute. The password itself is used as the cryptographic key for authentication and encryption. The *name* attribute contains the user name of the actor making the request. The *SecurityToken* element itself is empty.

7.1.2.4 Cleartext

This security mode is equivalent to passing the username and password in the clear and depends upon the use of a secure transport (such as SSL or IPSec). The purpose of including this security token type is to enable authentication to occur from web browsers over SSL or over internal LANs who use IPSec to encrypt all traffic. The password (or a hash of the password like in /etc/passwd) would have to be known by the server for authentication to occur. In this mode, neither encryption or signing of the hash is performed at the application layer. This mode is indicated by specifying a value of “Cleartext” for the *type* attribute. The *name* attribute contains the user name of the actor making the request and the string content of the *SecurityToken* element is the unencrypted plaintext password.

7.1.2.5 Kerberos

The use of a Kerberos version 5 token is indicated by specifying “Kerberos5” in the *type* attribute. The *name* attribute is used to contain the kerberos user id of the actor making the request. The *SecurityToken* element contains two subelements. The *Authenticator* element contains the authenticator encoded in base64. A *Ticket* element contains the service-granting ticket, also base64 encoded.

7.1.2.6 GSI (X.509)

The Grid Security Infrastructure (GSI) which is based on public key encryption, X.509 certificates, and the Secure Sockets Layer (SSL) communication protocol can be indicated by specifying a *type* attribute of "X509v3". The name attribute contains the userid used that the actor was mapped to in the local system. The string content of the *SecurityToken* element is the GSI authentication message including the X.509 identity of the sender encoded in base64.

7.1.3 Example

```
<SecurityToken type="Asymmetric" name="scottmo">
        MIIEZzCCA9CggAwIBAgIQEmtJZc0rqrKh5i...
</SecurityToken>
```

7.2 Authentication

Authentication entails how the peers at each end of the connection are identified and verified. Authentication is optional in an SSSRMAP message or reply. SSSRMAP uses a digital signature scheme for authentication that borrows from concepts in XML Digital Signatures [XML_DSIG]. In addition to authentication, the use of digital signatures also ensures integrity of the message, protecting exchanges from third-party modification.

When authentication is used, a *Signature* element is prepended as the first element within the *Envelope* element. All of the security modes will create a digest of the data for integrity checking and store this in base64 encoding in a *DigestValue* element as a child of the *Signature* element. The digital signature is created by encrypting the hash with the appropriate security token and storing this value in a *SignatureValue* element as a child of the *Signature* element. The security token itself is included as a child of the *Security* element within a *SecurityToken* element.

There are a number of procedural practices that must be followed in order to standardize this approach. The digest (or hash) is created over the contents of the *Envelope* element (not including the Element tag or its attributes). This might be over one or more *Request* or *Notify* elements (or *Response* or *Ack* elements) and necessarily excludes the *Signature* Element itself. (Note that any data encryption is performed after the creation of the digital signature and any decryption is performed before authenticating so the *EncryptedData* element will not interfere with this process. Hence, the signature is always based on the (hashed but) unencrypted data). For the purposes of generating the digest over the same value, it is assumed that the data is first canonicalized to remove extraneous whitespace, comments, etc according to the XML Digital Signature algorithm

("http://www.w3.org/TR/2001/REC-xml-c14n-20010315") and a transform is applied to remove namespace information. As a rule, any binary values are always transformed into their base64 encoded values when represented in XML.

7.2.1 The *Signature* Element

The *Signature* element MUST contain a *DigestValue* element that is used for integrity checking. It MUST also contain a *SecurityToken* element that is used to indicate the security mode and token type, and to verify the signature. It MUST contain a SignatureValue element that contains the base64 encrypted value of the signature wrought on the hash UNLESS the security token type indicates Cleartext mode where a signature would be of no value with the encryption key being sent in the clear -- in this case we use the password itself for authentication).

```
<complexType name="SignatureType">
  <choice minOccurs="2" maxOccurs="3">
    <element ref="sssrmap:DigestValue" minOccurs="1" maxOccurs="1"/>
    <element ref="sssrmap:SignatureValue" minOccurs="1" maxOccurs="1"/>
    <element ref="sssrmap:SecurityToken" minOccurs="0" maxOccurs="1"/>
  </choice>
</complexType>

<element name="Signature" type="sssrmap:SignatureType"/>
```

7.2.2 The *DigestValue* Element

The *DigestValue* element contains the cryptographic digest of the message data. As described above, the hash is generated over the *Body* element. The data to be hashed must first be canonicalized and appropriately transformed before generating the digest since typically an application will read in the XML document into an internal binary form, then marshal (or serialize) the data into a string which is passed as input to the hash algorithm. Different implementations marshal the data differently so it is necessary to convert this to a well-defined format before generating the digest or the clients will generate different digest values for the same XML. The SHA-1 [SHA-1] message digest algorithm (http://www.w3.org/2000/09/xmldsig#sha1) SHALL be used as the default method for generating the digest. A *method* attribute is defined as an extensibility option in case an implementation wants to be able to specify alternate message digest algorithms.

It MAY have a method attribute:

- *method* – the message digest algorithm.

- A *method* attribute of "sha1" specifies the SHA-1 message digest algorithm. This is the default and is implied if this attribute is omitted.

```
<complexType name="DigestValueType">
  <simpleContent>
    <extension base="string">
      <attribute name="method" type="string" use="optional"/>
    </extension>
  </simpleContent>
</complexType>

<element name="DigestValue" type="sssrmap:DigestValueType"/>
```

7.2.3 The *SignatureValue* Element

The SignatureValue element contains the digital signature that serves the authentication (and potentially non-repudiation) function. The string content of the SignatureValue element is a base64 encoding of the encrypted digest value. The HMAC algorithm [HMAC] based on the SHA1 message digest (http://www.w3.org/2000/09/xmldsig#hmac-sha1) SHALL be used as the default message authentication code algorithm for user identification and message integrity. A *method* attribute is defined as an extensibility option in case an implementation wants to be able to specify alternate digital signature algorithms.

It MAY have a method attribute:

- *method* – the digest signature algorithm.
  - A *method* attribute of "hmac-sha1" specifies the HMAC SHA-1 digital signature algorithm. This is the default and is implied if this attribute is omitted.

```
<complexType name="SignatureValueType">
  <simpleContent>
    <extension base="string">
      <attribute name="method" type="string" use="optional"/>
    </extension>
  </simpleContent>
</complexType>

<element name="SignatureValue" type="sssrmap:SignatureValueType"/>
```

7.2.4 Signature Example

Pre-authentication:

<Envelope>
    <Body>

```
                <Request action=”Query” actor=”kenneth”>
                        <Object>User</Object>
                        <Get name=”EmailAddress”></Get>
                        <Where name=”Name”>scott</Where>
                </Request>
        </Body>
</Envelope>
```

Post-authentication:

```
<Envelope>
        <Signature>
                <DigestValue>
                        LyLsF0Pi4wPU...
                </DigestValue>
                <SignatureValue>
                        DJbchm5gK...
                </SignatureValue>
                <SecurityToken type=”Asymmetric” name=”kenneth”>
                        MIIEZzCCA9CggAwIBAgIQEmtJZc0rqrKh5i...
                </SecurityToken>
        </Signature>
        <Body>
                <Request action=”Query” actor=”kenneth”>
                        <Object>User</Object>
                        <Get name=”EmailAddress”></Get>
                        <Where name=”Name”>scottmo</Where>
                </Request>
        </Body>
</Envelope>
```

7.3 Confidentiality

Confidentiality involves encrypting the sensitive data in the message, protecting exchanges against third-party interception and modification. Confidentiality is optional in an SSSRMAP message or reply. When confidentiality is required, SSSRMAP sessions use block cipher encryption with concepts borrowed from the emerging XML Encryption [XML_ENC] standard.

When confidentiality is used, encryption is performed over all child elements of the *Envelope* element, i.e. on the message data as well as any signature (The encrypted data is not signed -- rather the signature is encrypted). This data is replaced in-place within the envelope with an *EncryptedData* element. The data is first compressed using the gzip algorithm [ZIP]. Instead of encrypting this compressed data with the security token directly, a 192-bit random session key is

generated by the sender and used to perform symmetric encryption on the compressed data. This key is itself encrypted with the security token and included with the encrypted data as the value of the *EncryptedKey* element as a child of the *EncryptedData* element. The ciphertext resulting from the data being encrypted with the session key is passed as the value of a *CipherValue* element (also a child of the *EncryptedData* element). As in the case with authentication, the security token itself is included as a child of the *Security* element within a *SecurityToken* element.

7.3.1 The *EncryptedData* Element

When SSSRMAP confidentiality is required, the *EncryptedData* element MUST appear as the only child element in the *Envelope* element. It directly replaces the contents of these elements including the data and any digital signature. It MUST contain an *EncryptedKey* element that is used to encrypt the data. It MUST contain a *CipherValue* element that holds the base64 encoded ciphertext. It MAY also contain a *SecurityToken* element that is used to indicate the security mode and token type. If the *SecurityToken* element is omitted, a Symmetric key token type is assumed. Confidentiality is not used when a security token type of "Cleartext" is specified since it would be pointless to encrypt the data with the encryption key in the clear.

```
<complexType name="EncryptionDataType">
  <choice minOccurs="0" maxOccurs="1">
    <element ref="sssrmap:EncryptedKey" minOccurs="1" maxOccurs="1"/>
    <element ref="sssrmap:CipherValue" minOccurs="1" maxOccurs="1"/>
    <element ref="sssrmap:SecurityToken" minOccurs="1" maxOccurs="1"/>
  </choice>
</complexType>

<element name="EncryptedData" type="sssrmap:EncryptedDataType"/>
```

7.3.2 The *EncryptedKey* Element

The *EncryptedKey* element is a random session key encrypted with the security token. This approach is used for a couple of reasons. In the case where public key encryption is used, asymmetric encryption is much slower than symmetric encryption and it makes sense to use a symmetric key for encryption and pass along it along by encrypting it with the recipient's public key. It is also useful in that the security token which does not change very often (compared to the session key which changes for every connection) is used on a very small sampling of data (the session key), whereas if it was used to encrypt the whole message an attacker could more effectively exploit an attack against the ciphertext. The CMS Triple DES Key Wrap algorithm "kw-tripledes" SHALL be used as the default method for key encryption. The session key is encrypted using the security token, base64 encoded and specified as the string content of the *EncryptedKey* element. A

method attribute is defined as an extensibility option in case an implementation wants to be able to specify alternate key encryption algorithms.

It is REQUIRED that an implementation use a cryptographically secure Pseudo-Random number generator. It is RECOMMENDED that the session key be cryptographically generated (such as cyclic encryption, DES OFB, ANSI X9.17 PRNG, SHA1PRNG, or ANSI X12.17 (used by PGP)).

It MAY have a method attribute:

- *method* – the key encryption algorithm.
    - A *method* attribute of "kw-tripledes" specifies the CMS Triple DES Key Wrap algorithm. This algorithm is specified by the XML Encryption [XML_ENC] URI "http://www.w3.org/2001/04/xmlenc#kw-tripledes". It involves two Triple DES encryptions, a random and known Initialization Vector (IV) and a CMS key checksum. A 192-bit key encryption key is generated from the security token, lengthened as necessary by zero-padding. No additional padding is performed in the encryptions. This is the default and is implied if this attribute is omitted.

```
<complexType name="EncryptedKeyType">
  <simpleContent>
    <extension base="string">
      <attribute name="method" type="string" use="optional"/>
    </extension>
  </simpleContent>
</complexType>

<element name="EncryptedKey" type="sssrmap:EncryptedKeyType"/>
```

7.3.3 The *CipherValue* Element

The CipherValue element contains the message (and possibly signature) data encrypted with the random session key. The ciphertext is compressed using the gzip algorithm [ZIP], encrypted by the designated method, base64 encoded and included as the string content of the CipherValue element. The Triple DES algorithm with Cipher Block Chaining (CBC) feedback mode SHALL be used as the default method for encryption. A *method* attribute is defined as an extensibility option in case an implementation wants to be able to specify alternate data encryption algorithms.

It MAY have a method attribute:

- *method* – the data encryption algorithm.

- A *method* attribute of "tripledes-cbc" specifies the Triple DES algorithm with Cipher Block Chaining (CBC) feedback mode. This algorithm is specified by the XML Encryption [XML_ENC] URI identifier "http://www.w3.org/2001/04/xmlenc#tripledes-cbc". It specifies the use of a 192-bit encryption key and a 64-bit Initialization Vector (IV). Of the key bits, the first 64 are used in the first DES operation, the second 64 bits in the middle DES operation, and the third 64 bits in the last DES operation. The plaintext is first padded to a multiple of the block size (8 octets) using the padding scheme described in [XMLENC] for Block Encryption Algorithms (Padding per PKCS #5 will suffice for this). The resulting cipher text is prefixed by the IV. This is the default and is implied if this attribute is omitted.

```
<complexType name="CipherValueType">
  <simpleContent>
    <extension base="string">
      <attribute name="method" type="string" use="optional"/>
    </extension>
  </simpleContent>
</complexType>

<element name="CipherValue" type="sssrmap:CipherValueType"/>
```

7.3.4 Encryption Example

In this example, a simple request is demonstrated without a digital signature for the sake of emphasizing the encryption plaintext replacement.

Pre-encryption:

```
<Envelope>
        <Body>
                <Response>
                        <Status>true</Status>
                        <Code>000</Code>
                        <Count>1</Count>
                        <Data>
                                <User>

        <EmailAddress>Scott.Jackson@pnl.gov</EmailAddress>
                                </User>
                        </Data>
                </Response>
        </Body>
</Envelope>
```

Post-encryption:

```
<Envelope>
        <EncryptedData>
                <EncryptedKey>
                        NAkE9iQofYhyOfiHZ29kkEFVJ30CAwEAAaMSM...
                </EncryptedKey>
                <CipherValue>
                        mPCadVfOMx1NzDaKMHNgFkR9upTW4kgBxyPW...
                </CipherValue>
                <SecurityToken type=”Asymmetric” name=”kenneth”>
                        MIIEZzCCA9CggAwIBAgIQEmtJZc0rqrKh5i...
                </SecurityToken>
        </EncryptedData>
</Envelope>
```

8 Acknowledgements

9 References

[RFC2119] S. Bradner, “Key Words for Use in RFCs to Indicate Requirement Levels”, RFC 2119, March 1997.

[BEEP] M. Rose, “The Blocks Extensible Exchange Protocol Core”, RFC 3080, March 2001.

[HMAC] H. Krawczyk, M. Bellare, R. Canetti, “HMAC, Keyed-Hashing for Message Authentication”, RFC 2104, February 1997.

[SHA-1] U.S. Department of Commerce/National Institute of Standards and Technology, “Secure Hash Standard”, FIPS PUB 180-1.

[SSS] “Scalable Systems Software”, http://www.scidac.org/ScalableSystems

[HTTP] “Hypertext Transfer Protocol – HTTP/1.1”, RFC 2616, June 1999.

[XML_CONV] “I-X and <I-N-CA> XML Conventions”.

[FED_XML] “U.S. Federal XML Guidelines”.

[RFC3117] M. Rose, “On the Design of Application Protocols”, Informational RFC 3117, November 2001.

[XML_DSIG] D. Eastlake, J. Reagle Jr., D. Solo, "XML Signature Syntax and Processing", W3C Recommendation, 12 February 2002.

[XML_ENC] T. Imamura, B. Dillaway, E. Smon, "XML Encryption Syntax and Processing", W3C Candidate Recommendation, 4 March 2002.

[XRP] E. Brunner-Williams, A. Damaraju, N. Zhang, "Extensible Registry Protocol (XRP)", Internet Draft, expired August 2001.

[XML] Bray, T., et al, "Extensible Markup Language (XML) 1.0 (Second Edition)", 6 October 2000.

[XML_SCHEMA] D. Beech, M. Maloney, N. Mendelshohn, "XML Schema Part 1: Structures Working Draft", April 2000.

[ZIP] J. Gailly, M. Adler, "The gzip home page", http://www.gzip.org/